

Using HLS IP in a Zynq Soc Design

Lab #A- UG871

學號：R09921132

姓名：劉彥甫

高階技術合成於應用

2021年10月16日

Using HLS IP in a Zynq Soc Design	1
Lab #A- UG871	1
Vitis Development Flow	3
Pynq Z2 board settings	4
Lab 1- MAC design	5
System design in Vivado	10
Vitis application design	11
Lab 2 real2xfft	15
System design in vivado	18
Vitis application	19
The GitHub file hierarchy and note	20
Reference	21

Vitis Development Flow

In Lab A, we have to use the Vitis HLS, Vivado, and Vitis to finish the whole development flow. The Vitis HLS is the first step to design the algorithm we like to implement into configurable hardware such as FPGA. HLS also means fast prototyping, fast design space exploration, which can shorten the time to market. The second step is to use Vivado to perform system-level design, using the block diagram with on-the-shelf IPs to integrate a system. After complete the system design, we first need to generate the bitstream file and export the hardware platform file (.xsa). Reading the hardware platform file in Vitis can create a hardware platform including PS-PL communication protocols, hardware drivers. After creating the platform, we can start to build applications based on this hardware platform. This reference tutorial is based on Zynq 7000 boards, however, I port the flow onto Pynq z2 and successfully run the application in Vitis with actual FPGA.

Pynq Z2 board settings

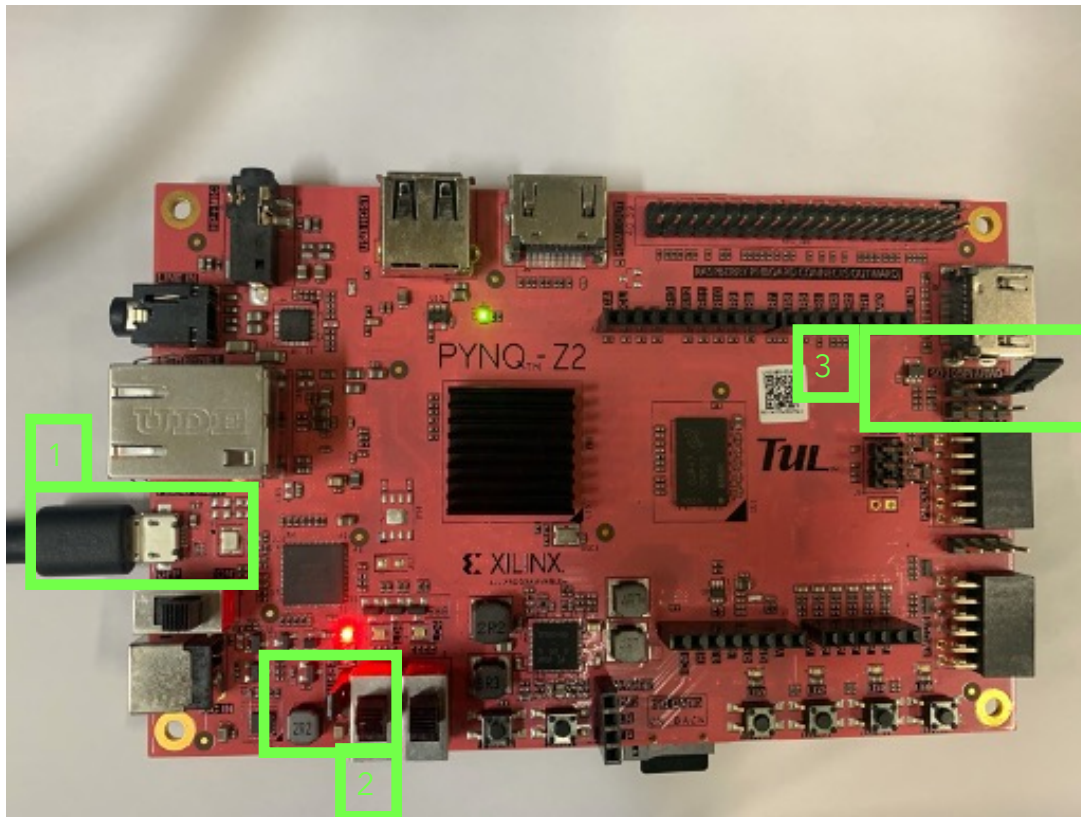


Fig. 1 Pynq Z2 Board top view.

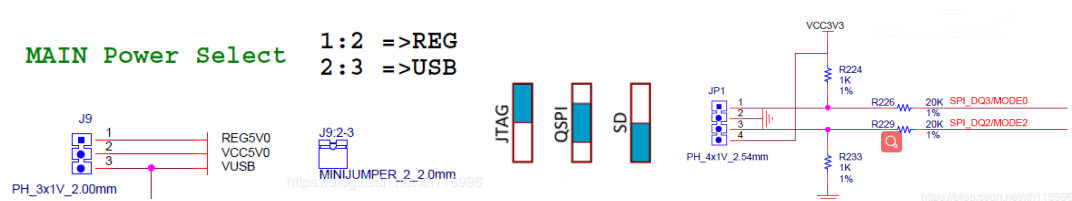


Fig. 2 J9 and J1 configuration

In Figure 1, the first rectangle is a micro USB port that needs to connect onto PC with the JTAG driver and the Vitis software installed; The second rectangle is a jumper J9, we need to short its 2, 3 pins with shorter as described in the left part of figure 2. The board will draw power from the USB port if the connection is valid. The third rectangle is a jumper J1 that we need to connect JTAG mode what makes FPGA can be programmed from JTAG.

Lab 1- MAC design

```
#include "hls_macc.h"

void hls_macc(int a, int b, int *accum, bool accum_clr)
{

#pragma HLS INTERFACE s_axilite port=return
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=a
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=b
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=accum
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=accum_clr
bundle=HLS_MACC_PERIPH_BUS

    static acc_reg = 0;
    if (accum_clr)
        acc_reg = 0;
    acc_reg += a * b;
    *accum = acc_reg;
}
```

Fig. 3 hls_macc.cpp

The original code is showed in figure 3, there are four I/O ports a, b, accum, and accum_clr.

```
void hls_macc(int a, int b, int *accum, bool accum_clr)
```

The port "a" and "b" is data with 32-bit data width and the port "accum" is the accumulated result(output) with 32-bit data width and the port accum_clr is one bit reset signal active at low(0).

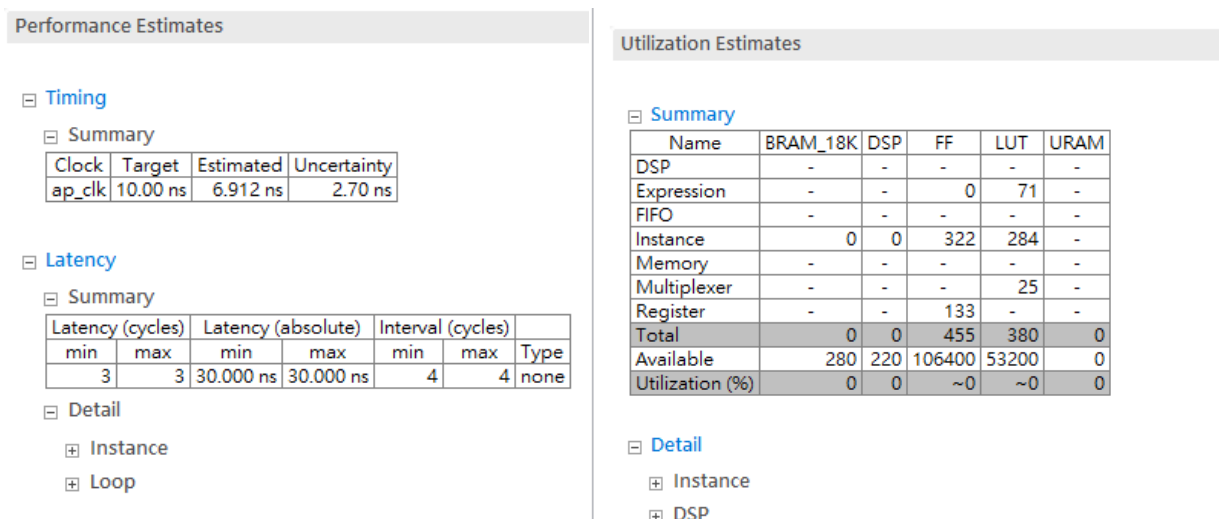
```
#pragma HLS INTERFACE s_axilite port=return
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=a
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=b
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=accum
bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=accum_clr
bundle=HLS_MACC_PERIPH_BUS
```

The interface is AXI-Lite and bundle with the same bus
"HLS_MACC_PERIPH_BUS".

The actual function part is quite simple, a register to store result and every

```
static acc_reg = 0;
if (accum_clr)
    acc_reg = 0;
acc_reg += a * b;
*accum = acc_reg;
```

time do current result = $a*b$ + previous result. We can see the timing performance is estimated 6.912 ns and only use FFs and LUTs to build the arithmetic operations.



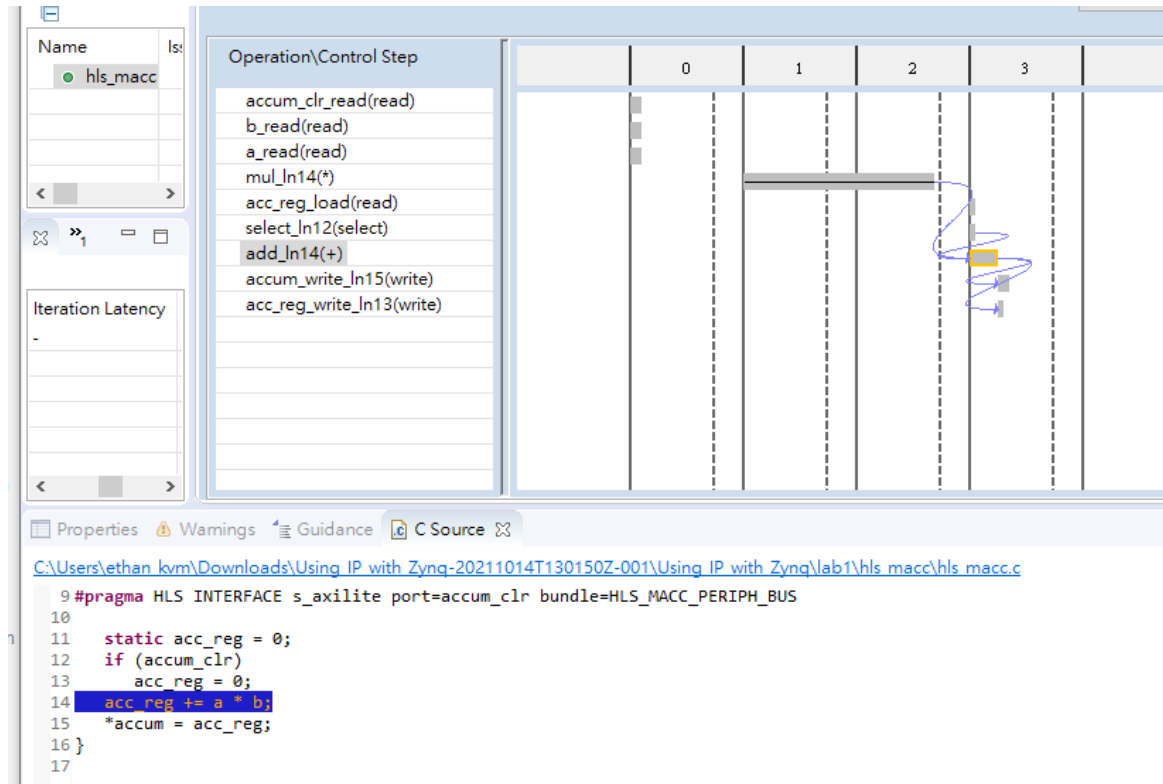


Fig. 5 Timing analysis

In figure 5, the function first loads the operand and performs multiplication and addition. Multiplication takes most of the run-time, maybe there is one trick we can do like using DSP resources. We can use DSP to do the addition and multiplication, but we need some pragma to guide the tool to use those hardware resources.

```

set_directive_top -name hls_macc "hls_macc"
set_directive_bind_op -op add -impl dsp "hls_macc" acc_reg
set_directive_bind_op -op mul -impl dsp "hls_macc" dsp

```

Fig. 6 Directive.tcl

```

static acc_reg = 0;
int dsp;
if (accum_clr)
    acc_reg = 0;
dsp = a * b;
acc_reg += dsp;
*accum = acc_reg;

```

Fig. 7 modified hls_macc.cpp

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.598 ns	2.70 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
3	3	30.000 ns	30.000 ns	4	4	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	32	-
FIFO	-	-	-	-	-
Instance	0	1	191	281	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	25	-
Register	-	-	133	-	-
Total	0	1	324	338	0
Available	280	220	106400	53200	0
Utilization (%)	0	~0	~0	~0	0

Fig. 8 modified version performance and utilization estimates

In the C synthesis, the actual DSP was used for multiplication. The number of FFs and LUTs is reduced and one DSP is used on the modified version. Not only the hardware resource utilization is changed, but the timing performance is also changed down to 6.598 ns. The main reason why the HLS tool doesn't use DSP is that the timing constraint is not tight in the original design. Since the original code uses pure LUT core, in the following session, I stick to the original version to perform the demonstration.

```
// RTL Simulation : 4378 / 4383 [0.00%] @ "2583115000"
// RTL Simulation : 4379 / 4383 [0.00%] @ "2583705000"
// RTL Simulation : 4380 / 4383 [0.00%] @ "2584295000"
// RTL Simulation : 4381 / 4383 [0.00%] @ "2584885000"
// RTL Simulation : 4382 / 4383 [0.00%] @ "2585475000"
// RTL Simulation : 4383 / 4383 [100.00%] @ "2586065000"
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
$finish called at time : 2586105 ns : File "C:/Users/ethan_kvm/AppData/Roaming/Xilinx/Vitis/use_ip_on
run: Time (s): cpu = 00:00:35 ; elapsed = 00:00:39 . Memory (MB): peak = 954.398 ; gain = 0.000
## quit
INFO: [Common 17-206] Exiting xsim at Thu Oct 14 21:15:48 2021...
INFO: [COSIM 212-316] Starting C post checking ...

*** 32 Tests Passed ***
```

Fig. 9 co sim passed

After co-sim passed, we can see the waveform in the Vitis tool. In figure 10, the waveform starts working when ap_start = 1 as we discussed in the previous lecture. It start with 0 and a = -1, b = -1. a*b = 1 and the first result is 0 + 1 = 1. The waveform confirms that it is the correct operation. After co-sim is done, we may export the RTL IP and move on to the next step.

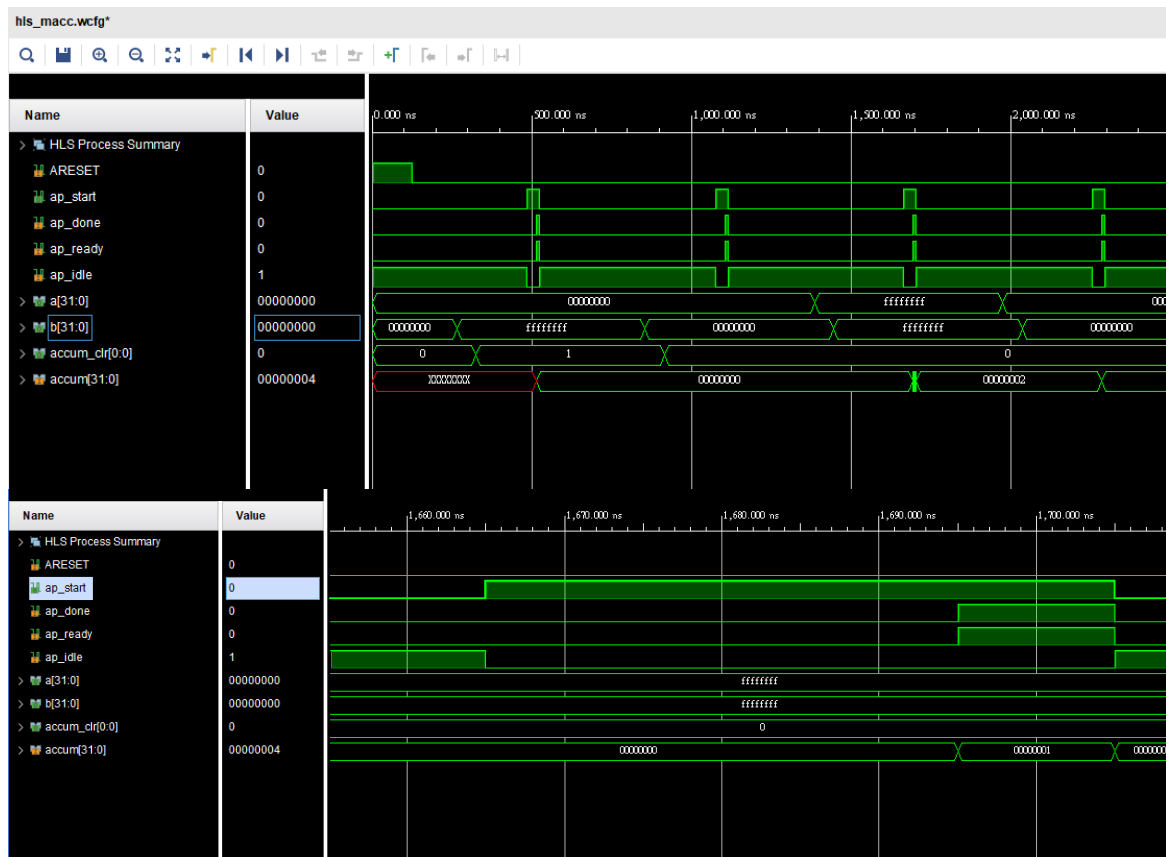


Fig. 10 Waveform dumped

System design in Vivado

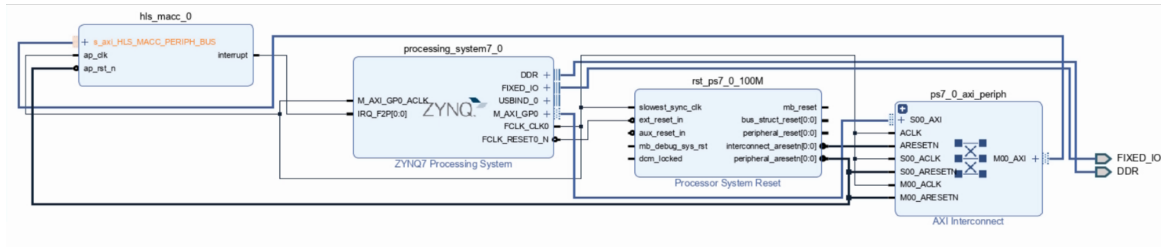


Fig. 10 Block design diagram

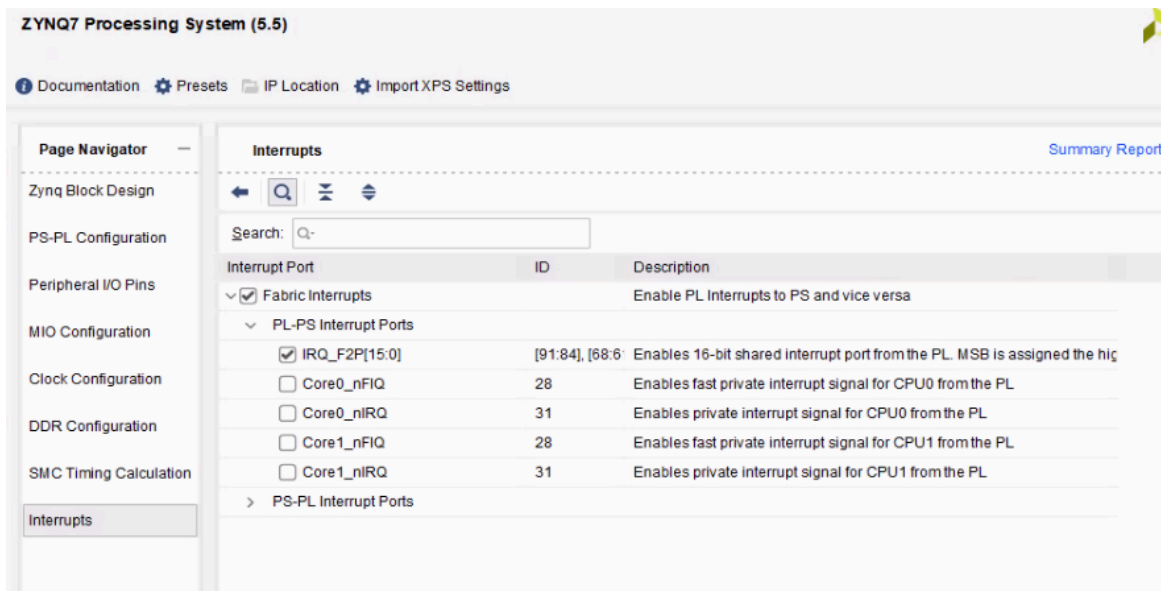


Fig. 11 ZYNQ7 config

The only thing new is the "Interrupt" signal which connects to PS IRQ(interrupt request). After creating ZYNQ 7 processing system, double-click the block and turn on IRQ_F2P[15:0] in figure 11. Connect the hls_macc_0 port "interrupt" and the ZYNQ7 port "IRQ_F2P[15:0]". The interrupt function can be used later in the Vitis session.

The final step in Vivado is to generate a bitstream file and export hardware into a (.xsa) file. This xsa file can be used to create a hardware platform in the Vitis.

Vitis application design

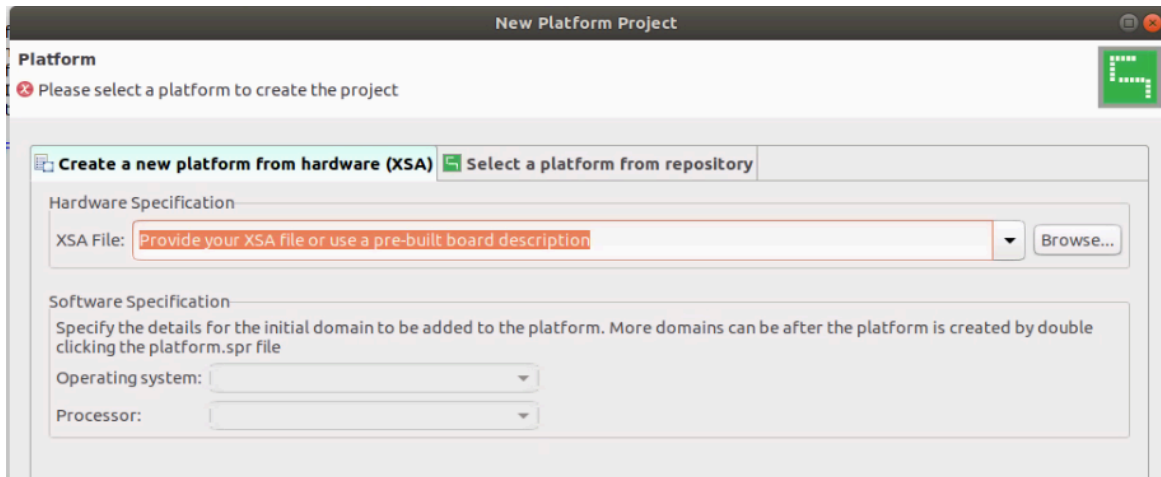


Fig. 12 New platform project UI

The first thing to do is to create a hardware platform through xsa file.

In figure 12, choose xsa file and create a hardware platform. After creating the hardware platform, we can create a hello world application to validate the hardware platform is solid.

```
void print(char *str);

int main()
{
    init_platform();

    print("Hello World\n\r");

    return 0;
}
```

Fig. 13 helloworld.c template code

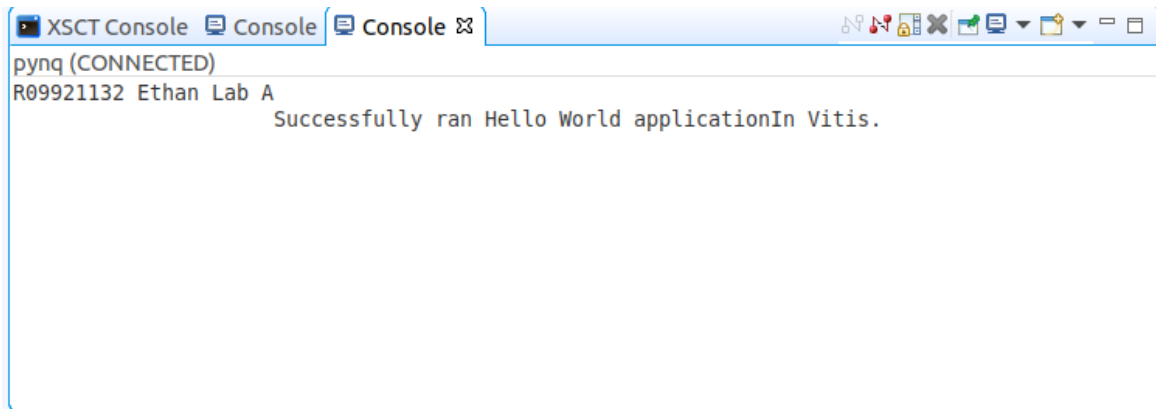


Fig. 14 The output result

In figure 14, the terminal in this version 2020.2 is kind of buggy. However, it still works and displays the context. You must received the hello word string to confirm the hardware platform is correct and solid. The next step is to run real code.

```
int main()
{
    print("Program to test communication with HLS MACC
block in PL\n\r");
    int a = 10, b = 21;
    int res_hw;
    int res_sw;
    int i;
    int status;

    //Setup the matrix mult
    status = hls_macc_init(&HlsMacc);
    if(status != XST_SUCCESS){
        print("HLS peripheral setup failed\n\r");
        exit(-1);
    }
    //Setup the interrupt
    status = setup_interrupt();
    if(status != XST_SUCCESS){
        print("Interrupt setup failed\n\r");
        exit(-1); }
```

```

//set the input parameters of the HLS block
XHls_macc_Set_a(&HlsMacc, a);
XHls_macc_Set_b(&HlsMacc, b);
XHls_macc_Set_accum_clr(&HlsMacc, 1);

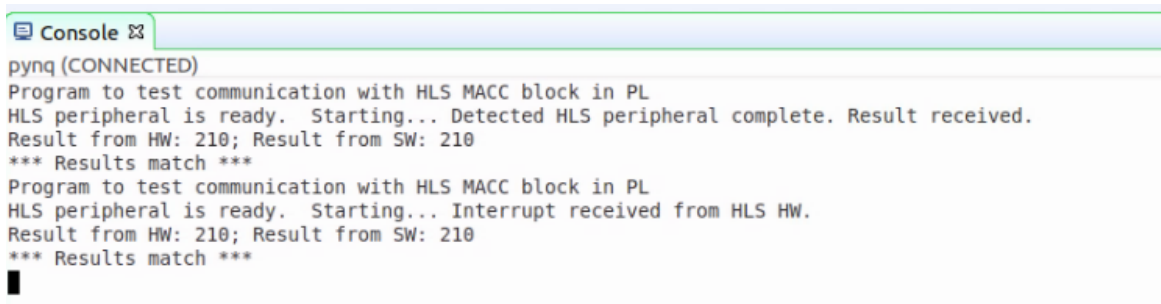
if (XHls_macc_IsReady(&HlsMacc))
    print("HLS peripheral is ready. Starting... ");
else {
    print("!!! HLS peripheral is not ready! Exiting...
\n\r");
    exit(-1);
}

if (1) { // use interrupt
    hls_macc_start(&HlsMacc);
    while(!ResultAvailHlsMacc)
        ; // spin
    res_hw = XHls_macc_Get_accum(&HlsMacc);
    print("Interrupt received from HLS HW.\n\r");
} else { // Simple non-interrupt driven test
    XHls_macc_Start(&HlsMacc);
    do {
        res_hw = XHls_macc_Get_accum(&HlsMacc);
    } while (!XHls_macc_IsReady(&HlsMacc));
    print("Detected HLS peripheral complete. Result
received.\n\r");
}

```

Fig. 15 macc code

In figure 15, you can see the interrupt code is to wait for hardware interrupt and get a result after. Also can use a non-interrupt mode, software poll the ready handshake signal. If the ready signal is high, means the result is ready and valid.



```
pynq (CONNECTED)
Program to test communication with HLS MACC block in PL
HLS peripheral is ready. Starting... Detected HLS peripheral complete. Result received.
Result from HW: 210; Result from SW: 210
*** Results match ***
Program to test communication with HLS MACC block in PL
HLS peripheral is ready. Starting... Interrupt received from HLS HW.
Result from HW: 210; Result from SW: 210
*** Results match ***
```

Fig. 16 Result console

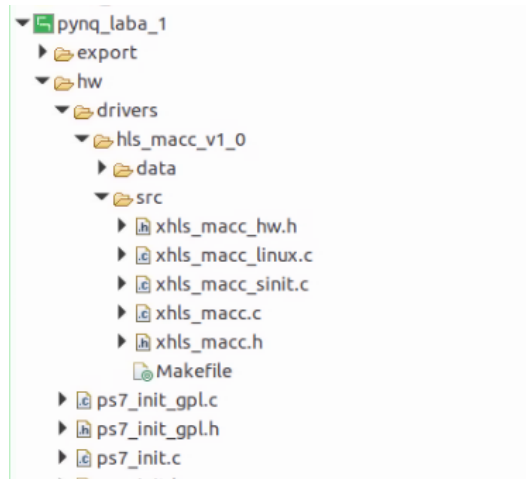


Fig. 17 hardware driver

If use non-interrupt mode, it is shown in the upper part of figure 16; If use interrupts mode, it is shown in the lower part of figure 16.

Here are some notes, the driver call in the application, you have to check the hardware platform driver part as shown in figure 17. Otherwise, the application won't be compiled without errors. You also have to include the driver header file in the application in order to fetch the corresponding function.

Lab 2 real2xfft

The HLS code was divided into 2 parts, the first part is to prepare the sliding window which contains N, number of samples, sample with half new and half old samples; the second part is to prepare the output data into streaming data out. The original code doesn't contain any FFT algorithm like the radix-2 butterfly algorithm.

The important thing is on the 68 line of the reference code, the pragma is no longer support in this version 2020.2.

```
#pragma HLS ARRAY_PARTITION variable=nodelay,delayed  
cyclic factor=2
```

Fig. 18 bad pragma

After co-sim pass, you have to export IP on both front-end function and back-end function in figure 19.

```
void hls_real2xfft(  
    hls::stream<din_t>& din,  
    hls::stream<xfft_axis_t<dout_t> >& dout)  
{  
    #pragma HLS INTERFACE axis port=dout  
    #pragma HLS INTERFACE axis port=din  
        din_t data2window[REAL_FFT_LEN],  
    windowed[REAL_FFT_LEN];  
    #pragma HLS ARRAY_PARTITION variable=data2window cyclic  
    factor=2  
    #pragma HLS ARRAY_PARTITION variable=windowed cyclic  
    factor=2  
    #pragma HLS ARRAY_STREAM variable=data2window,windowed  
    depth=2  
    #pragma HLS DATAFLOW
```

Fig. 19 FE/BE code

```

    sliding_win_1in2out<din_t, REAL_FFT_LEN>(din,
data2window);
    window_fn<din_t, din_t, coeff_t, REAL_FFT_LEN,
WIN_FN_TYPE, 2>(
        data2window, windowed);

real2xfft_output:
    for (int i = 0; i < REAL_FFT_LEN; i += 2) {
#pragma HLS PIPELINE rewind
        complex<dout_t> cdata(windowed[i], windowed[i + 1]);
        xfft_axis_t<dout_t> fft_axis_d;
        fft_axis_d.data = cdata;
        fft_axis_d.last = i == REAL_FFT_LEN - 2 ? 1 : 0;
        dout.write(fft_axis_d);
    }
}
// BACK END
#include "xfft2real.h"

// This is the top-level (for HLS) function for the Real
FFT backend processing
void hls_xfft2real(
    hls::stream<xfft_axis_t<dout_t> >& din,
    hls::stream<xfft_axis_t<dout_t> >& dout)
{
#pragma HLS INTERFACE axis port=dout
#pragma HLS INTERFACE axis port=din
#pragma HLS DATA_PACK variable=dout
#pragma HLS DATAFLOW

    // Template functions cannot be the top-level for
HLS...
    xfft2real<dout_t, dout_t, LOG2_REAL_FFT_LEN, true>(din,
dout);
}

```

Fig. 19 FE/BE code

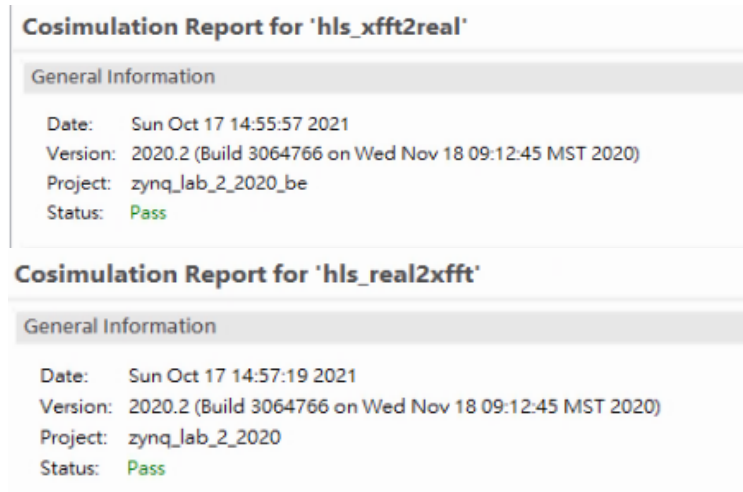


Fig. 19-1 co-sim result

In figure 19-1, The co-sim pass on both FE and BE IP and the performance and utilization estimates is in figure 19-2 and 19-3.

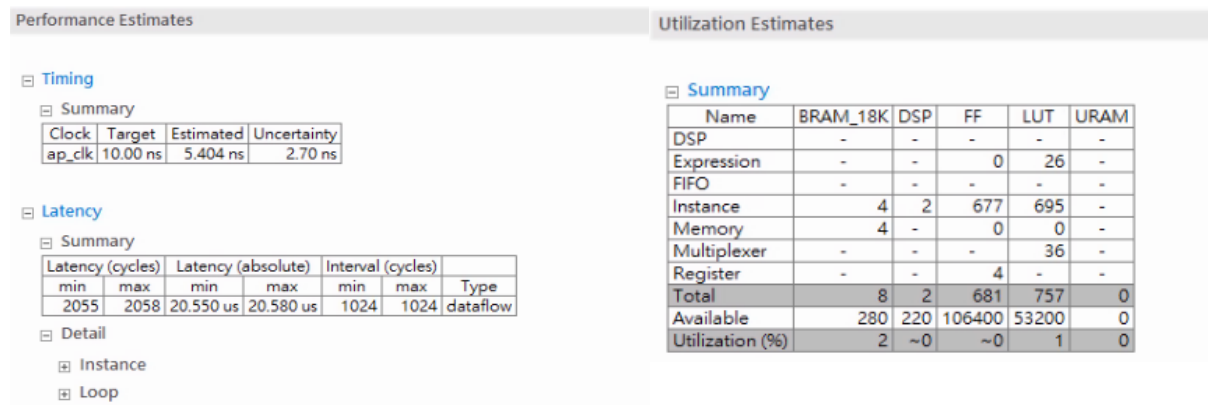


Fig. 19-2 FE Estimates

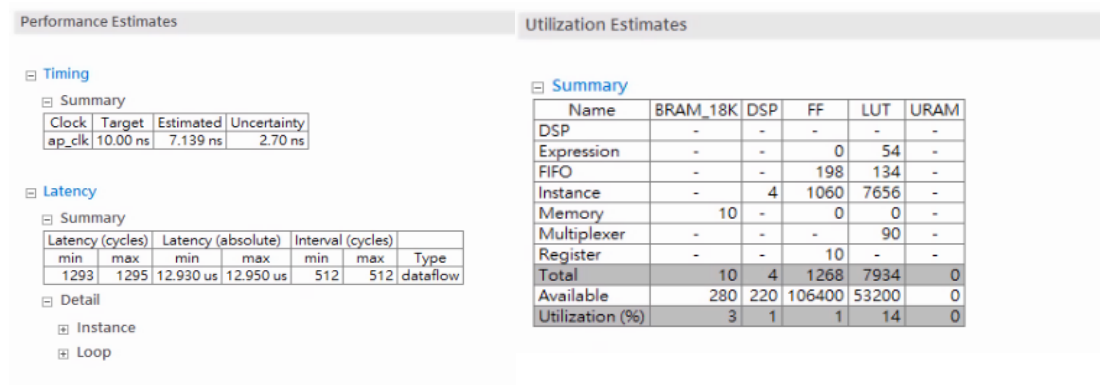


Fig. 19-3 BE Estimates

System design in vivado

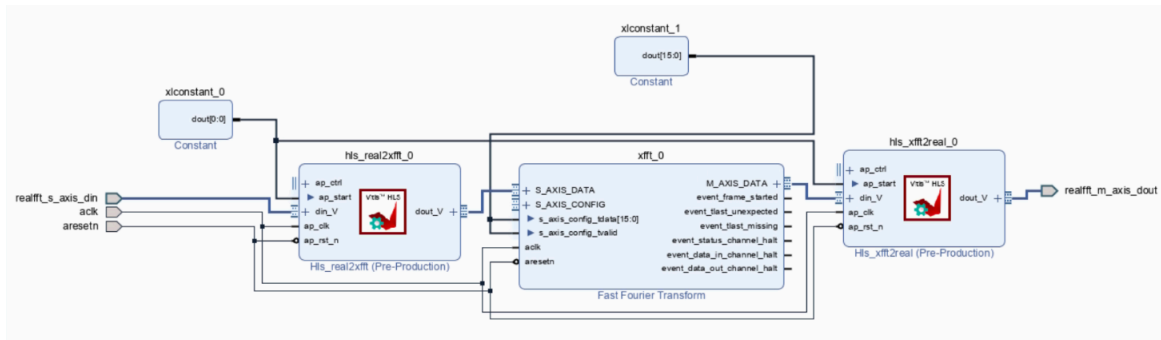


Fig. 20 RealFFT block diagram

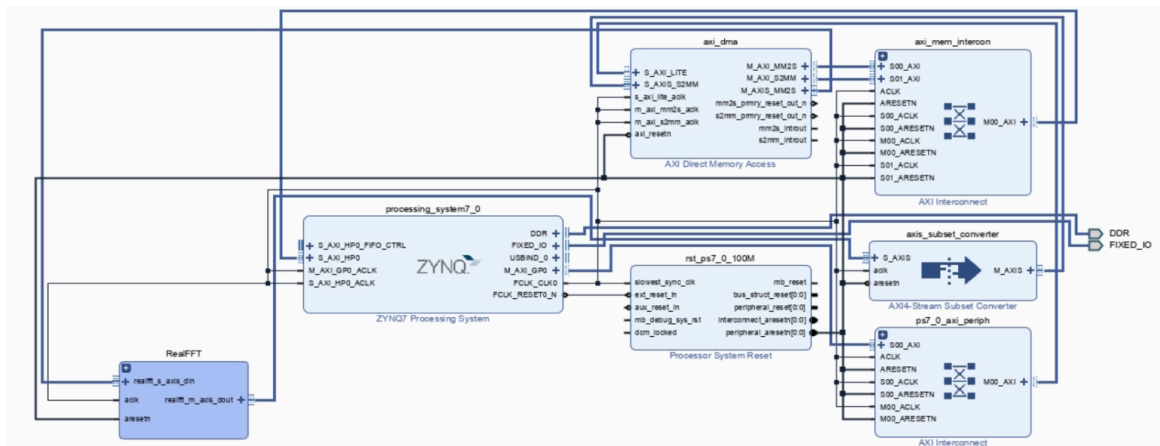


Fig. 21 Top level block diagram

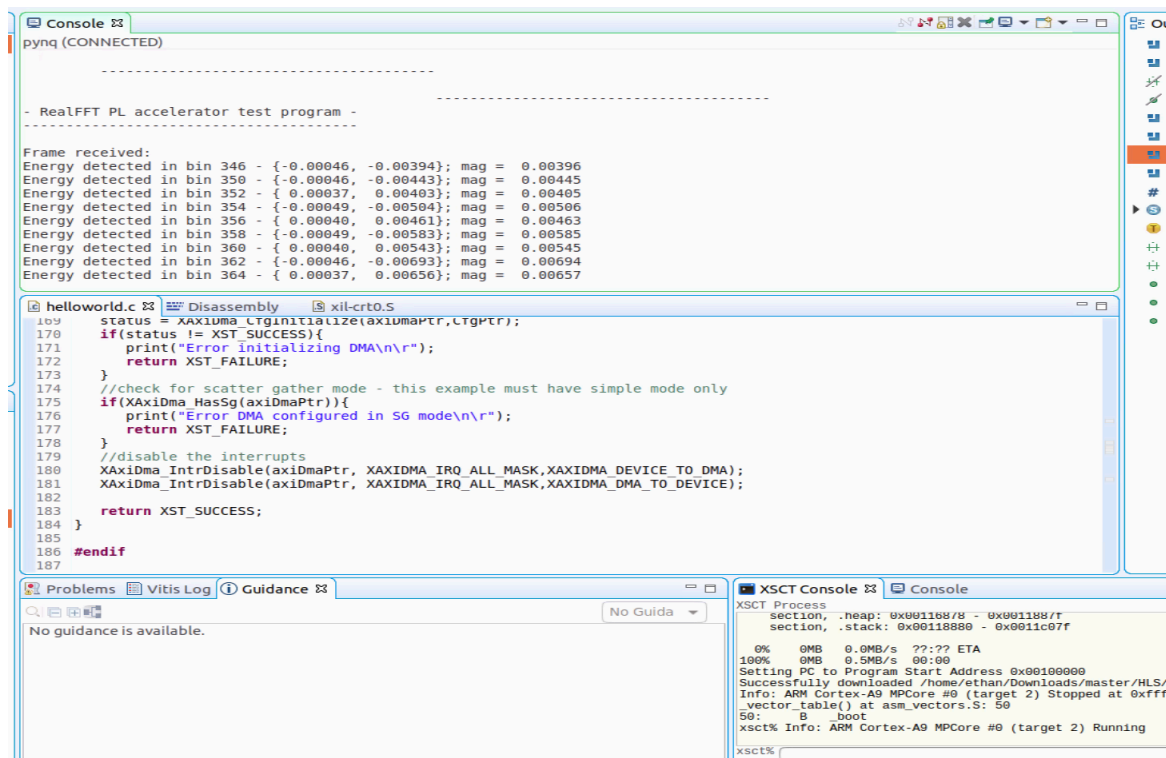
In figure 20, add an on-the-shelf IP "fast Fourier transform" and the front-end IP read2xfft, sliding window function, and the backend IP xfft2real. There are some constant blocks that need to be connected on both ap_start and on the AXI-Stream config of FFT IP.

This block diagram cannot be done by the automation connection in the Vivado tool, it must be done by manual connection. One trick to make sure the block diagram is correct is to right-click on the block diagram canvas and chose the valid design, if the result is shown without a critical error message, it should be fine.

After generating the bitstream file, the last step before closing the Vivado is to export the xsa file.

Vitis application

The Vitis application is first to generate a waveform for input data on PS side and then send the sample to the first FE IP for the window sliding function. The data is normalized with 1-bit integer and 15-bit fractions. The FE IP will send the data into FFT IP to perform the Fourier transform. Once the computation is done, the result will be sent back to xfft2real IP and then we can retrieve the data. If data magnitude is greater than a threshold, we will print it out as shown in figure 22.



```
Console
pynq (CONNECTED)

-----
- RealFFT PL accelerator test program -
-----

Frame received:
Energy detected in bin 346 - {-0.00046, -0.00394}; mag = 0.00396
Energy detected in bin 350 - {-0.00046, -0.00443}; mag = 0.00445
Energy detected in bin 352 - { 0.00037, 0.00403}; mag = 0.00405
Energy detected in bin 354 - {-0.00049, -0.00504}; mag = 0.00506
Energy detected in bin 356 - { 0.00040, 0.00461}; mag = 0.00463
Energy detected in bin 358 - {-0.00049, -0.00583}; mag = 0.00585
Energy detected in bin 360 - { 0.00040, 0.00543}; mag = 0.00545
Energy detected in bin 362 - {-0.00046, -0.00693}; mag = 0.00694
Energy detected in bin 364 - { 0.00037, 0.00656}; mag = 0.00657

helloworld.c  Disassembly  xil-crt0.S
169 status = XAXIDMA_CfgInitialize(axiDmaPtr, CfgPtr);
170 if(status != XST_SUCCESS){
171     print("Error initializing DMA\n\r");
172     return XST_FAILURE;
173 }
174 //check for scatter gather mode - this example must have simple mode only
175 if(XAXIDMA_HasSg(axiDmaPtr)){
176     print("Error DMA configured in SG mode\n\r");
177     return XST_FAILURE;
178 }
179 //disable the interrupts
180 XAXIDMA_IntrDisable(axiDmaPtr, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DEVICE_TO_DMA);
181 XAXIDMA_IntrDisable(axiDmaPtr, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DMA_TO_DEVICE);
182
183 return XST_SUCCESS;
184 }
185 #endif
186
187

Problems  Vitis Log  Guidance
No guidance is available.

XSCT Console  Console
XSCT Process
section, .heap: 0x001188/8 - 0x001188/r
section, .stack: 0x00118880 - 0x0011c07f
0% 0MB 0.0MB/s ???:?? ETA
100% 0MB 0.5MB/s 00:00
Setting PC to Program Start Address 0x00100000
Successfully downloaded /home/ethan/Downloads/master/HLS/
Info: ARM Cortex-A9 MPCore #0 (target 2) Stopped at 0xffff
vector_table() at asm_vectors.S: 50
50: 0 _boot
xsct% Info: ARM Cortex-A9 MPCore #0 (target 2) Running
xsct%
```

Fig. 22 Vitis result

The GitHub file hierarchy and note

The GitHub link is : [https://github.com/Waxpple/Using-HLS-IP-in-a-Zynq-](https://github.com/Waxpple/Using-HLS-IP-in-a-Zynq-Soc-Design)

Soc-Design

The file hierarchy is:

```
├─ Lab1
│   ├── vitis
│   │   ├── hello
│   │   ├── hello_system
│   │   └─ pynq_laba_1
│   ├── vitis_hls
│   │   └─ use_ip_on_zynq_lab1
│   └─ vivado
│       └─ hls_ip_zynq_lab1
├─ Lab2
│   ├── vitis
│   │   ├── hello_system_2
│   │   ├── hello_system_2_system
│   │   └─ pynq_lab2
│   ├── vitis_hls
│   │   ├── zynq_lab_2_2020
│   │   └─ zynq_lab_2_2020_be
│   └─ vivado
│       └─ hls_ip_zynq_lab2
├─ Reference_code
│   └─ Using_IP_with_Zynq
│       ├── lab1
│       └─ lab2
└─ Slide_report
    ├── Lab_A_R09921132_劉彥甫.pdf
    └─ Lab_A_R09921132_劉彥甫.pptx
```

24 directories, 2 files

Reference

[1] Xilinx, "ug871-vivado-high-level-synthesis-tutorial", https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug871-vivado-high-level-synthesis-tutorial.pdf